USING PREDICTIVE MODELS FOR COMPRESSION IN DATABASE SYSTEMS

Aswin Karumbunathan
Undergraduate Thesis
Computer Science, Sc. B.
Brown University 2013

Advisor: Ugur Cetintemel
Reader: Tim Kraska

Abstract

Lossy compression in databases can yield better compression ratios than lossless compression in general, yet is rarely used due to the concern of losing data. However, with the rise of big data, lossy compression has become acceptable for many uses, analytics databases being one of them. Semantic compression, one form of lossy compression, uses associations within the data to compress the data rather than blindly compressing the database as if it was one large file. In column-wise semantic compression, columns are compressed based on their associations with other columns by using predictive models. These predictive models could be used for more than just bulk decompression, if the query engine knows of the models. The models have an idea of the distribution of the data, which the query engine is free to use. Columns can be selectively decompressed, and a graph between models can be created to model the depth of each model, or how many other models it depends on. In this way, models can be adjusted in a way such that the most commonly used columns can be kept uncompressed, and the least used columns can be kept compressed more deeply than the other columns. A proof-of-concept program was used to test these notions, but future work involves implementing this in a real database system to see how this works in practice.

CONTENTS

1 INTRODUCTION

Compression yields the obvious improvement of reducing the amount of storage space required. Compression can also improve performance by reducing the amount of IO required to fetch data from storage, whether it be disk IO or network IO. Today, network and disk IO tend to be bottlenecks for most computer systems far more often than CPU limitations, given how CPU performance has grown at a much faster rate than network or disk transfer rates. Lossless compression is the more commonly chosen type of compression, for the sake of not losing data. However, now that massive amounts of data are being stored, regardless of their usefulness, using lossy compression seems like a reasonable trade-off, to sacrifice some accuracy in return for improved compression and performance. An example of a particular scenario where this applies is a database used for analytics, where only approximate values are required.

Semantic compression, a type of lossy compression, is a relatively newer idea. Semantic compression uses relationships within the data to compress it, rather than regarding the database as a file that needs to be compressed. Semantic compression can be done in a row-based manner, which involves clustering together rows with a bound amount of variation, or a column-based manner, which involves removing columns that can be predicted using other columns and storing any exceptions that exceed a certain error bound separately. While row-wise semantic compression in general shows better compression ratios, column-wise compression has its own advantages. Here these advantages are covered, along with some experimental results using a prototype compression/decompression program.

2 RELATED WORK ON DATABASE COMPRESSION

Different lossless compression methods have been used for databases. The use of `gzip` and other traditional lossless compression methods in databases is fairly straightforward. More specialized lossless compression techniques can be used, however. Abadi et al. [1] use different lossless compression methods on a column-by-column basis, using methods such as dictionary encoding and run-length encoding. However, these methods were implemented in C-Store, which is a column-oriented DBMS. While using a system such as C-Store could certainly make column-wise semantic compression easier to use, the methods described there do not adapt easily to row-oriented DBMS as well (run-length encoding, for example).

Fascicles [5] is one of the earlier methods developed for semantic compression. This method performs row-wise compression, finding groups of rows that have a limited amount of variation. Spartan [3] combines both of the standard semantic compression approaches, first classifying columns using decision trees then clustering the resulting rows to achieve greater compression. Spartan was shown to achieve better compression ratios than `gzip` and fascicles on selected datasets. ItCompress [6] takes the approach of Fascicles but improves the algorithm and method enough (applying the row-wise compression algorithm iteratively) that it achieves better compression ratios than even Spartan. While ItCompress achieves a better compression ratio than Spartan, Spartan's usage of compression in a more transparent manner makes it a more attractive choice for integration into a widely used DBMS. This transparency can be useful, as the query engine can take these into account when running queries, and using these cleverly could yield significant performance improvements.

The basic model for column-wise compression used by Spartan is the same one used here. First, columns that can be compressed must be found somehow; Spartan uses Bayesian networks to achieve this. Next, decision trees are built for every column that can be compressed. To improve performance, these trees can be built using some representative sample of the data rather than the entire dataset. Finally, the number of columns in the dataset is compressed, keeping track of any exceptions that exceed the error threshold, so that all that needs to be stored are the models, the uncompressed columns, and the exceptions.

3 COMPRESSION

Column-wise compression can be applied to database systems fairly simply. A predictive model needs to be trained to classify the column to be compressed. Once the model is trained, it then needs to be applied to the column in question. If the model shows a high enough predictive accuracy, then that column can be completely removed from the database, otherwise the column can be kept entirely and the model can be discarded. Error bounds can be placed to limit the loss from this lossy compression, on both numerical values and nominal values. Error bounds on numerical values can be kept by calculating how different the predicted value is from the original value. Of course, there is no such metric for nominal values, so instead the error bound can be used to not compress columns whose predictive accuracy has an error greater than that error bound. So error bounds are applied on a per-case basis for numerical values and on a per-column basis for nominal values.

For both numeric and nominal columns, exception lists can be stored in order to help achieve compression for attributes whose predictive models do not have the accuracy required to compress a particular column. When a column is being compressed, any values for whom the predictive model has too high of an error can have their original values stored elsewhere, so that the predictive model can use that value instead of the predicted value when that particular row is decompressed. This way, the error bound is maintained yet the column can still be compressed, while maintaining some bounds on the total error that can be observed. Instead of classifying nominal values on a per-column basis, they could be compressed by simply storing exceptions for any invalid predictions, although this may increase the total storage cost if there are many invalid predictions.

However, there is a potential problem here. The error bound is maintained for each column, but if one predictive model depends on another predictive model, it is possible for errors to propagate between models. Figure 1 shows an example of this. This could cause inaccuracies greater than the guaranteed error bound. This problem is not easily surmountable, although there are various workarounds that could help alleviate the issue. One possible way to work around this is to make sure to train the predictive models on what was compressed of the dataset so far, forcing the new models to be built and error-checked using the values of the predictive models it depends on, if any. However, this would increase the performance cost of compression significantly, either requiring the intermediate compressed values to be stored or for the decompressed values to be rederived at every step of compression. Another possible workaround may be to limit the depth of the models, so that errors can only be propagated so far, although this can only have limited impact.

The prototype shows that compression results are actually quite decent for some datasets, given that only off-the-shelf classifiers were used, rather than specialized models (as in Spartan [3]). One limitation of column-wise semantic compression is that compression ratios may vary greatly from dataset to dataset, with some datasets showing high levels of compression and others showing none or nearly none. Another limitation is that, to be integrated into a DBMS, a sample of the data must be available before anything is compressed. Since the format of the table is not being changed,

Row	Visitors	Clicks	Subscribes
1	1000	500	51
2	1200	600	60
3	1400	700	65
4	2000	950	90
5	2400	1600	162
6	3300	2200	220
7	3500	2300	233

Subscribes	**Clicks**
: Clicks $*$ 0.1	*Visitors* $\leq$ 2400 : Visitors $*$ 0.5
	Visitors $>$ 2400 : Visitors $*$ 0.667

Applying these rules to row 3, where both Clicks and Subscribes are compressed:

Visitors = 2000 $\rightarrow$ Clicks = 1000 $\rightarrow$ Subscribes = 100

The Subscribes column now exceeds the error bound of (90 $*$ .95, 90 $*$ 1.05), although either model used on its own does not cause this issue.

Figure 1: Example of error propagation.

at least from the user's point of view, the entire storage process could happen offline, to improve performance, rather than happening synchronously.

It is interesting to note that the order that the columns are compressed in shows remarkably little effect on the compression ratio achieved. It could have been possible that in order to compress some columns certain other columns would have to be uncompressed. However, the associations between the columns tend to work out in a way such that the order in which compression is done does not matter. Empirical evidence shows that although there is some variation when the order of compression is varied, the overall compression ratio does not change by much.

4 DECOMPRESSION

Column-wise compression differs from most other methods of compression in the fact that the entire dataset, or an entire row, does not have to be decompressed at once in order to use the dataset. If only uncompressed columns need to be used, then no decompression is necessary. If compressed columns do need to be used, only the columns that need to be used and any other columns required by the predictive models for the desired columns need to be decompressed. The speed gained from this would help performance, even though decompression is relatively fast in the first place, as the prototype was able to show. Decompression took a matter of seconds for most of the datasets used, although a significant portion of that time is due to the latency of reading from and writing to the file system one entry at a time, which is simply a side-effect of how the program works. Decompression is far faster than compression, which is an acceptable trade-off for most database uses, especially if offline compression is done. If custom decision trees were built, it would be possible to create a compiled version of them, so to speak, for the sole purpose of decompression, especially since the decision trees are equivalent to a series of if statements.

One advantage of using decision trees is that these models are transparent and can be understood by the query engine. Figure 2 shows a few example decision trees, which are easily understandable at first glance. The decision trees below are the string representation of the decision tree objects. If a query engine with knowledge of Soil Type 25's decision tree was asked a simple SELECT query with a WHERE clause like this (WHERE Soil Type 25 = 1), then it could instantly respond to that query with a blank response, since it knows that there are no cases where Soil Type 25 is equal to 1, at least within the error bound currently in use. Each decision tree also holds distribution information about the column it is predicting. The numbers in the brackets and parenthesis for each line show the number of correctly and incorrectly classified instances for that leaf from the training set (parenthesis) and the pruning set (brackets). The query engine could use this distribution information when comparing different query execution plans. For example, the query engine can know that Wilderness Area 1 has a value of 1 for the majority of rows, which could help when planning a query with a join, for example. The decision tree stores these values to calculate information gain, which is what it is maximizing, but it turns out that it could also be helpful for the query engine.

REPTree for Wilderness Area 1

```
Model training instances = 4,999

Wilderness Area 3 = 0
|   Wilderness Area 4 = 0
|   |   Wilderness Area 2 = 0 : 1 (3548/0) [1775/0]
|   |   Wilderness Area 2 = 1 : 0 (21/0) [19/0]
|   Wilderness Area 4 = 1 : 0 (181/0) [79/0]
Wilderness Area 3 = 1 : 0 (249/0) [127/0]
```

REPTree for Soil Type 25

```
Model training instances = 4,999

: 0 (3999/0) [2000/0]
```

Figure 2: Example of some decision trees from the Forest Cover dataset. The numbers around the slashes denote the number of correctly/incorrectly classified attributes for the training set (parenthesis) and the pruning set (brackets)

The query engine can learn the associations between the models and decide which models need to be decompressed for each particular query. An easy way to represent this is in a graph, where nodes represent columns and have directed edges to nodes that are required for their predictive models. In such a graph, nodes with no outgoing edges have no dependencies, and are either uncompressed or can be decompressed directly. All other nodes have some number of other columns that they depend on, and accessing them will have a higher performance cost. This graph can be used by the query engine to represent costs for decompressing each attribute that it can take into account when planning out query execution.

In addition to being used in query integration, the graph of dependencies can also be changed depending on which columns are used most frequently. The most accessed columns should be kept uncompressed, since that will be faster. The least accessed columns can be left at the top of the model dependency graph, where they will depend on many other columns and models. The depth of predictive models can be controlled by limiting the columns that the predictive model can use. This way, the costs for each column can be modified in a way that is proportional to the amount each column is used. As stated earlier, the order of compression does not affect the final compression ratio by much, so reordering the predictive models should not reduce the amount of compression that can be achieved, unless of course some columns are kept uncompressed for speed.

6 PROTOTYPE AND RESULTS

6.1 *Datasets*

FOREST-COVER (`kdd.ics.uci.edu/databases/covertype/`) This dataset contains forest cover data for 30 x 30 meter cells obtained from US Forest Service (USFS) Region 2 Resource Information System (RIS) data. It contains 10 numeric attributes, 44 binary attributes, and 1 nominal attribute.

COREL (`kdd.ics.uci.edu/databases/CorelFeatures/`) This dataset contains color histograms for image features extracted from a Corel image collection. It contains an ID attribute and 32 numeric attributes.

US CENSUS BUREAU (`http://www.census.gov/cps/data/`) This dataset, obtained from the DataFerret tool, contains household, demographic, and geographic data from March 2013. It contains 2 ID attributes and 49 numeric attributes. Some of the numeric attributes could be treated as nominal attributes, as they have a finite range of values that correspond to some predefined set of values. For example, a numeric attribute is used to represent states, rather than strings. All attributes are treated as numeric attributes here.

SDSS (`http://www.sdss.org/`) This Sloan Digital Sky Survey dataset was obtained from surveys of the sky using a dedicated 2.5-meter telescope at Apache Point Observatory, New Mexico that collected a wide variety of measurements. It consists of an ID attribute and 508 numeric attributes. Only a subset of the data was used here, rather than the entire dataset, which is enormous.

TPC-H (`http://www.tpc.org/tpch/`) The traditional database benchmark, the TPC-H dataset, was also used here, although only the parts table was used. The parts table contains an ID attribute, a string attribute, 2 numeric attributes, and 5 nominal attributes.

6.2 *Compression Prototype*

The program created to test these hypotheses and run experiments uses classification algorithms from Weka and MOA, which are described further below and have their source code freely available online. As depicted in Figure 3, it takes as input either a dataset to compress in ARFF format, or a compressed dataset to be decompressed. The ARFF file format, used for input, is simply a CSV with a header that details the type of each attribute and also details for nominal attributes all the possible values. The classification packages will continue to work even as new values to nominal attributes are added, although old values may not be removed, so this is not a problem for attributes that may have new categories added. The output format for compression, which may be input to the program to be decompressed, is a folder containing a CSV file with empty values for compressed attributes, two header files, and the serialized decision tree models for all compressed columns. The program's output format can be input back into the program and vice versa.

Compression is achieved one column at a time. Models are trained on samples of the data and the error for every column is approximated on a

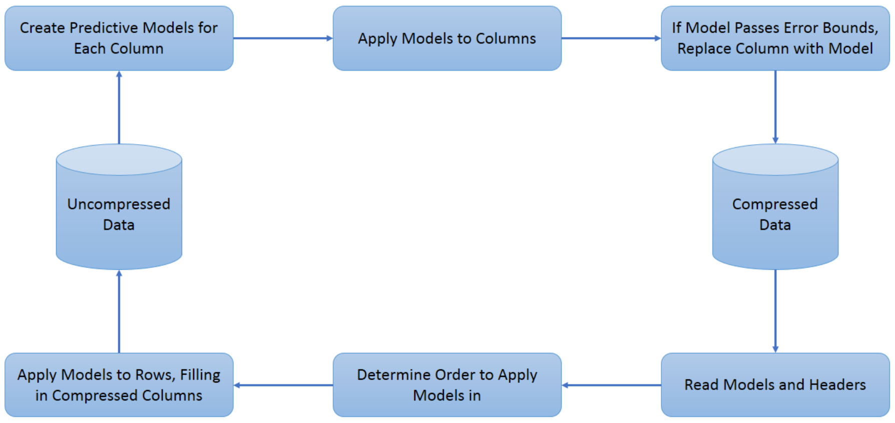
Create Predictive Models for Each Column
Apply Models to Columns
If Model Passes Error Bounds, Replace Column with Model
Uncompressed Data
Compressed Data
Apply Models to Rows, Filling in Compressed Columns
Determine Order to Apply Models in
Read Models and Headers

Figure 3: Diagram of basic flow of prototype program.

larger sample of the data. The columns are then compressed individually in order from the column with the least error to the column with the most error. First, a model is trained on a sample of the data, then applied to the column. If the error is low enough, then the column is compressed and all values of the column that fit within the requested error bound are removed. Every time a column is compressed, its values are no longer available for later models to use when being trained. The models are serialized and output along with the final data file.

Decompression is done in the opposite manner. The models, which are not very large, are read into memory, then they are applied to each row in the dataset in the opposite order they were applied in, to ensure that all the columns the model depends on have been decompressed if necessary.

6.3 *Predictive Models*

Classifier models from the Weka package (`http://www.cs.waikato.ac.nz/ml/weka/`) and from the MOA package (`http://moa.cms.waikato.ac.nz/details/`) were used. The MOA package, an extension to Weka adapted for on-line data, was used to make the program handle streaming data. Not only is required to handle larger data sets, it also presents a more realistic usage. Weka classifiers require the whole dataset to be given to the classifier before it is constructed and are clearly impractical for any real usage in a database system without the MOA extension.

The specific classifier models used from these packages were decision trees, in specific the M5P, REPTree, and Hoeffding Tree classifiers. These are all decision trees designed to be built quickly, at the cost of having a decision tree that is not minimal. The Hoeffding Tree classifier (from MOA) is a decision tree model that can be built incrementally. The REPTree classifier (from Weka) is a decision tree classifier designed for speed, using reduced error pruning. The M5P classifier (from Weka) builds a model tree for numeric attributes that can outperform REPTree in some cases in terms of accuracy. MOA provides a wrapper class that trains WEKA models incrementally, which was used for all of the Weka models used. All three implementations can be converted into a direct representation of the tree easily. That is, the internals of the decision tree models can be viewed to see how the tree was built.

Decision tree models were chosen over other types of classifiers, such as neural networks and regression models, for their speed, transparency, and ability to find complex relationships in data with multiple columns. Neural networks are much slower than either of the decision tree models presented above. They also function as black boxes, not allowing any insight into the associations derived by the networks. Regression models are easier to interpret, but are not as good at finding complex associations in data with multiple columns where there may be associations between multiple attributes.

6.4 *Compression Ratios*

Figure 4 shows the compression ratios achieved across the five datasets used as the error bound was varied from 0 to 10%. Compression ratio is defined as the size of the compressed dataset divided by the size of the uncompressed dataset, so lower compression ratios are better. The Census dataset only showed notable compression ratios upon relaxation of the error bound beyond zero, as did the Corel and SDSS datasets, and these three datasets show improved compression ratios as the error bound is relaxed further. The Forest Cover dataset shows good compression ratios from the beginning, and those improve as the error bound is relaxed, although by less with each step. Increasing the error bound from 5% to 10% shows little increase in compression ratio. Finally, the parts table from the TPC-H dataset shows constant compression, because only one column was able to be compressed, and that was compressed with 0% error. These different datasets are quite diverse, and it is easily seen that semantic compression can perform better on some datasets than others.

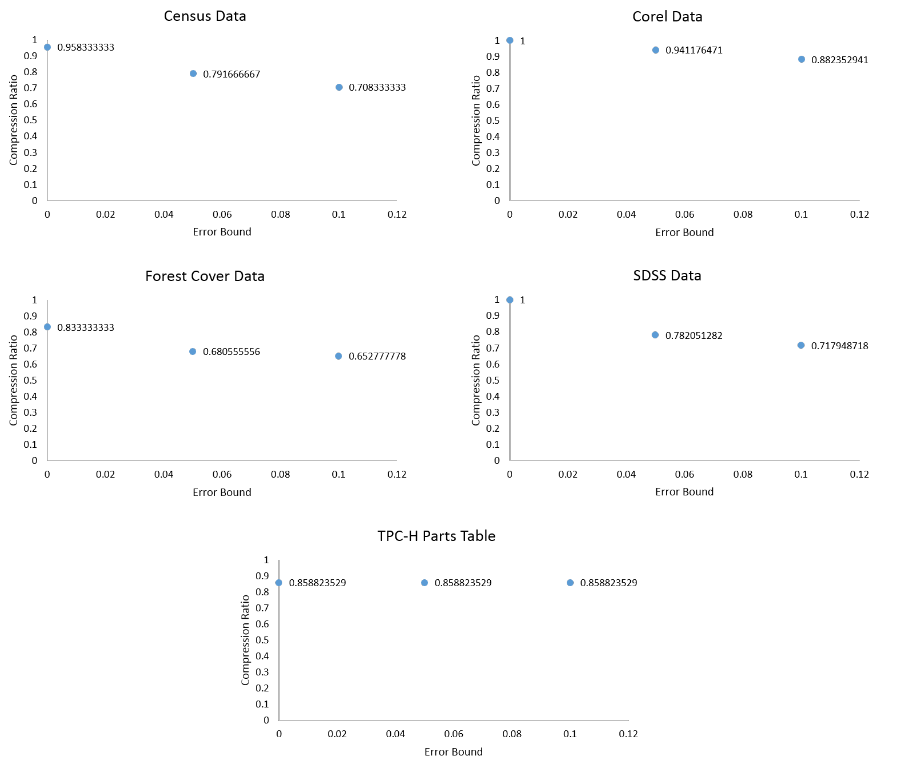

Figure 4: Compression ratios across the five datasets used for different error bounds.

- Compression ratio = (size of compressed database) / (size of uncompressed database)

6.5 *Compression Order*

Figure 5 plots histograms of the number of columns compressed to certain extents. This way, different orders of compression can be compared easily to see how well each of them compresses the dataset. In all cases, there are no significant effects on compression from the order chosen. Three different strategies were used to order columns for compression: compressing the columns whose model showed the least error first, compressing columns randomly, and compressing columns in increasing/decreasing order of model complexity, where complexity is defined as the number of other columns a model depends on. None of these three strategies seemed to significantly improve the compression ratios achieved.

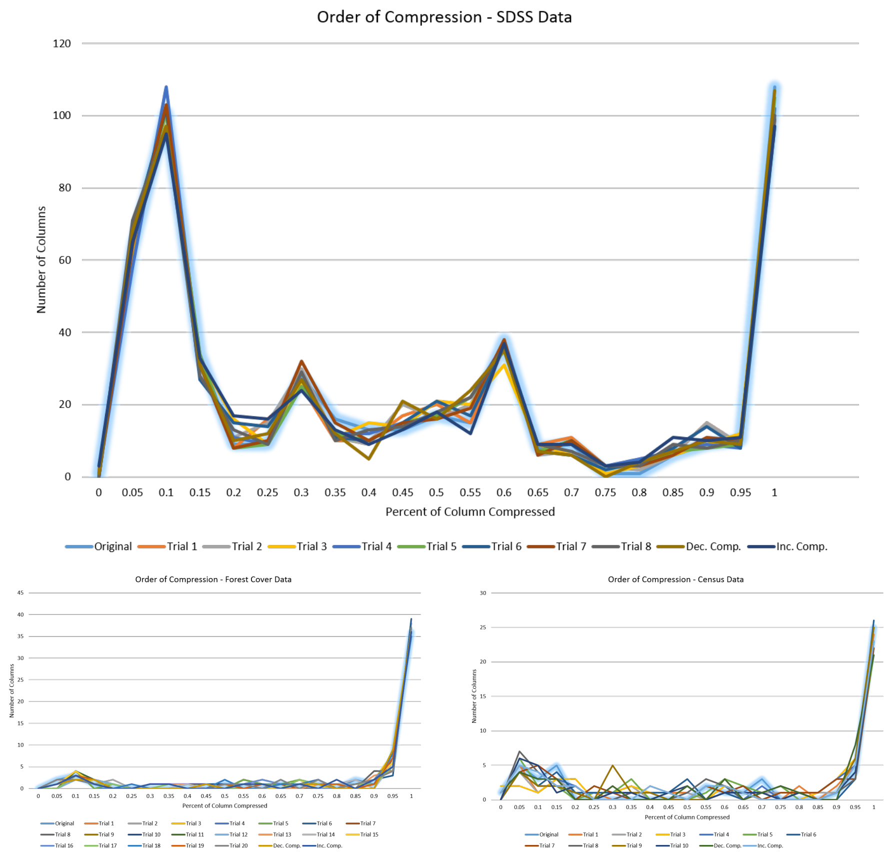

Figure 5: Histogram of per-column compression for trials compressing in different orders, plotted as lines for easy comparison of different trials. Definition of trial types:

- Original (highlighted): columns compressed in order of decreasing predictive accuracy
- Trial #: columns compressed in random order
- Inc./Dec. Comp.: columns compressed in order of increasing/decreasing complexity (where complexity is defined as the number of models the predictive model is dependent on)

7 CONCLUSION

Column-wise semantic compression can be achieved in a straightforward manner, and shows decent results even using off-the-shelf predictive models. This method shows more promise in its ability to use these predictive models to help the query engine make more educated decisions when planning execution and calculating execution costs. If statistics are collected about how often columns are used, the models can be tweaked to match the workload as well. Since the models themselves contain information about the distribution of the data, this could also be used when designing query execution plans. Despite only requiring a light level of integration, this method can adapt remarkably well and achieve notable performance improvements. However, semantic compression is limited by the dataset being compressed, as some datasets may not show relationships between columns, although this tends to be the exception rather than the common case.

8 FUTURE WORK

8.1 *Topics to Investigate*

The effect of compression order on the final compression ratios appeared minimal, but the experiments run here could be supplemented. A trial using a subset of data that is known to be strongly associated may help solidify this claim. If the order of compression within this subset has no effect on the final compression ratios, then it is safer to conclude that the order of compression really does have no effect on compression ratios. Further modeling and investigation into how to control the effects of error propagation may also prove to be helpful, as well as other possible ways to integrate query workload data into this scheme.

8.2 *Possible Improvements*

The prototype used here reflects the straightforward implementation of column-wise semantic compression, but there are a number of places where improvements could be used. These improvements could boost the advantage of this form of compression relative to traditional compression methods or row-wise compression methods, although of course there is nothing stopping using these forms of compression together. Using specialized models would serve to improve compression ratios and could decrease decompression speeds. If models were able to predict the values of multiple columns at once, then that could improve both accuracy and speed. Since many columns could be related with one another, having models that could predict the values of multiple attributes could predict multiple columns at a time, where before only one could be predicted at a time.

Methods described from previous papers on database compression could also be applied here. BlinkDB [2], a system that creates multiple samples to allow users to specify accuracy or time constraints on queries, uses stratified sampling to build its samples. Stratified sampling is used in place of uniform sampling to ensure that a sample of the database contains values even from the tails of the distribution. This could be used to train predictive models here. This especially applies to numerical attributes, given that a model has no way to predict a nominal value unless it has been trained on at least one row with that value. Another method that could be used during query execution was developed by Chen et al., called transient decompression. The standard methods of decompression are eager decompression, where the data is decompressed as soon as it is brought into memory, or lazy decompression, where data is decompressed as required. Transient decompression decompresses the data as it is being operated on, but keeps the compressed version of the data and passes that along to the next operator instead of the decompressed version. This was shown to improve performance for hashing and sorting operations, and could also be useful here.

8.3 *Database Implementation*

The next step for future work is to implement simple column-wise semantic compression in a database. It is possible to envision a naive user-level implementation of this, which would involve simply by creating a function that uses a predictive model to generate values for a column that has been removed from a table. In Hive, this would correspond to a map task fairly simply. Of course, this would require work on the user's part to create and use these models, but part of the attraction of column-wise semantic compression is how easy it seems to integrate it into a database system, even one that is not column-oriented, so implementing this by integrating it into a database system is the more helpful task, which should also give an idea of how well this idea works in practice.

When implementing semantic compression in a database system, things to keep note of would include: the amount of work required to transparently integrate this into the query engine, the ability to adapt compression to changing workloads, and the performance improvements achieved from making these changes. These changes could be implemented in a standard database system, such as Postgres, or a distributed database system, such as Hive. Implementing these in a column-based database system, such as C-Store, could also prove interesting, but since column-based database systems are not widely used, this may be less useful.

9 ACKNOWLEDGEMENTS

I would first like to thank Professor Ugur Cetintemel for all his help and for being my Thesis Advisor. I would also like to thank Professor Tim Kraska for being the second reader for this thesis, and Kayhan Dursun for the helpful discussions.

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682. ACM, 2006.

[2] Sameer Agarwal, Aurojit Panda, Barzan Mozafari, Samuel Madden, and Ion Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. *arXiv preprint arXiv:1203.5485*, 2012.

[3] Shivnath Babu, Minos Garofalakis, and Rajeev Rastogi. Spartan: A model-based semantic compression system for massive data tables. *ACM SIGMOD Record*, 30(2):283–294, 2001.

[4] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. In *ACM SIGMOD Record*, volume 30, pages 271–282. ACM, 2001.

[5] HV Jagadish, Jason Madar, and Raymond Ng. Semantic compression and pattern extraction with fascicles. In *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*, pages 186–197, 1999.

[6] HV Jagadish, Raymond T Ng, Beng Chin Ooi, and Anthony KH Tung. Itcompress: An iterative semantic compression algorithm. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 646–657. IEEE, 2004.